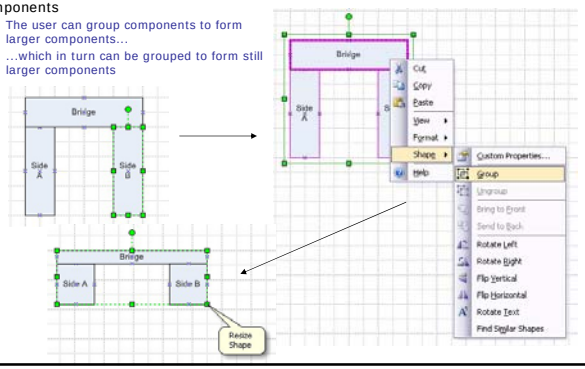
ECE450 – Software Engineering II

Today: **Design Patterns VI**

Composite, Iterator, and Visitor Patterns

Think of drawing/diagramming editors

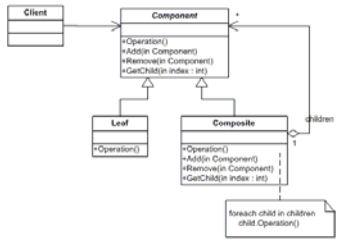
- Drawing/diagramming editors let users build complex diagrams out of simple components
  - The user can group components to form larger components…
  - …which in turn can be grouped to form still larger components



The problem

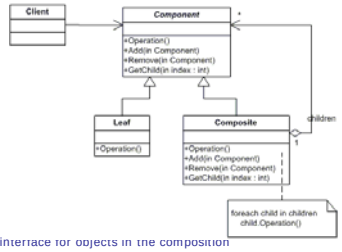
- Although we grouped the shapes, the system should not treat the result as a "primitive" shape
  - Even if most of the time we treat them identically
  - We may decide to ungroup them
    - Or to modify an attribute of one of the components of the grouped shape
- The *Composite* pattern describes how to use recursive composition so that clients don't have to make this distinction.

The *Composite* pattern



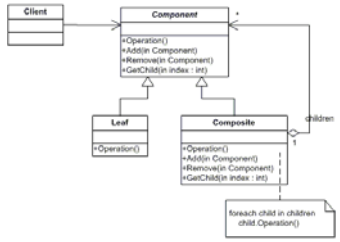
- ***Composite**: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly*
  - The key to the pattern is the abstract class that represents *both* primitives and their containers.

Structure and participants



- Component
  - Declares the interface for objects in the composition
  - Implements default behavior for the interface common to all classes
  - Declares an interface for accessing and managing its child components
  - (Optionally) defines an interface for accessing a component's parent in the structure, and implements it if appropriate
- Leaf
  - Represents leaf objects in the composition. A leaf has no children
  - Defines behavior for primitive objects in the composition

Structure and participants (cont)



- Composite
  - Defines behavior for components having children
  - Stores child components
  - Implements child-related operations in the Component interface
- Client
  - Manipulates objects in the composition through the Component interface

Applicability

- Use the Composite pattern when…
  - You want to represent part-whole hierarchies of objects
  - You want clients to be able to ignore the difference between compositions of objects and individual objects.
    - Clients will treat all objects in the composite structure uniformly
- A second example: file systems
  - Directories contain entries, each of which could be a directory

Consequences

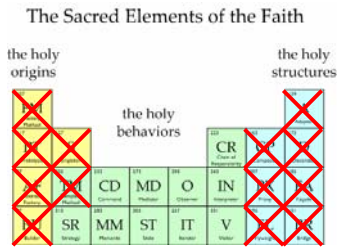
- The pattern defines class hierarchies consisting of primitive objects and composite objects
  - Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively.
  - Wherever client code expects a primitive object, it can also take a composite object
- Makes the client simple
  - Clients can treat composite structures and individual objects uniformly.
  - Clients don't know (and shouldn't care) whether they're dealing with a leaf or a composite component
- But can make your design overly general
  - The disadvantage of making it easy to add new components is that it makes it harder to *restrict* the components of a composite
    - Sometimes you want only some *kinds* of components, but you'll have to use run-time checks to guarantee your constraints.

Implementation

- Explicit parent references?
  - You may want to maintain references from child components to their parent to simplify traversal and management of a composite structure
  - The usual place to define the parent reference is in the Component class
- Declaring the child management operations?
  - Our diagram has the Component abstract class declare the Add and Remove operations for managing children, but leaf nodes never use them
  - Is it better to have those operations declared in the Composite class?
  - Tradeoff between safety and transparency:
    - Defining the child management interface at the root gives you transparency (you can treat all components uniformly) but costs you safety (clients may try to do meaningless things like add and remove objects from leaves!)
    - Defining child management in the Composite class gives you safety (any attempt to add or remove objects from leaves will be caught at compile-time), but you lose transparency (leaves and composites have different interfaces).
- What is the best data structure for storing components?
  - Depends on efficiency. It is possible that you don't even need a general-purpose data structure at all!

That was it for structurals... Now we need to tackle behaviorals

- We've gone through the creational patterns...
  - Factory method
  - Abstract factory
  - Builder
  - Prototype
  - Singleton
- ... the structural ones...
  - Decorator
  - Adapter
  - Facade
  - Proxy
  - Flyweight
  - Bridge
- ...and two behaviorals
  - Template method
  - Null object



- A (short) review is in order...

Behavioral patterns

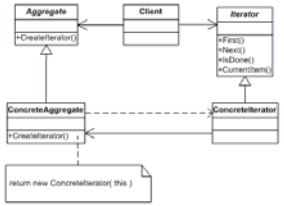
- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects
  - They describe not just patterns of objects or classes, but also the patterns of *communication* between them

We'll start with a simple one...

- Assume you have an aggregate object such as a list
- This aggregate object should give you a way to access its elements without exposing its internal structure.
  - And you may want to traverse the list in different ways
- However, adding the operations for different traversals to the List interface would bloat it
  - The List is there for a purpose *different than* supporting our traversals
    - And remember classes should do just **one** thing, if possible (maximizing cohesion)
  - Not only that, but you may want to traverse it more than once at the same time
- How can we address this problem?

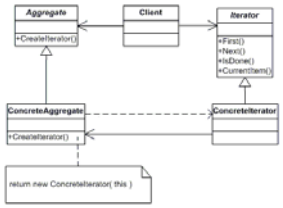
The *Iterator* pattern

- ***Iterator:*** *Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation*
  - You probably knew of this one already
    - Java's Iterator interface is an application of this principle



Structure and participants

- Iterator
  - Defines an interface for accessing and traversing elements
- ConcreteIterator
  - Implements the Iterator interface
  - Keeps track of the current position in the traversal of the aggregate
- Aggregate
  - Defines an interface for creating an Iterator object
- ConcreteAggregate
  - Implements the Iterator creation interface to return an instance of the proper ConcreteIterator



Applicability

- Use the Iterator pattern…
  - To access an aggregate object's contents without exposing its internal representation
  - To support multiple traversals of aggregate objects
  - To provide a uniform interface for traversing different aggregate structures
    - That is, to support polymorphic iteration

Consequences

- Iterators support variations in the traversal of an aggregate
  - Complex aggregates may be traversed in many ways. Iterator subclasses (or Iterator instances) allow clients to traverse the aggregate in the way they desire
- More than one traversal can be pending on an aggregate
  - An iterator keeps track of its own traversal state. Therefore, you can have more than one traversal in progress at once

Implementation

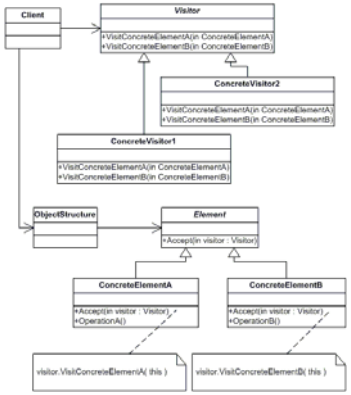
- Who controls the iteration?
  - When the client controls the iteration, the iterator is called an **external iterator**
    - Clients must advance the traversal and request the next element explicitly
  - When the iterator controls it, the iterator is an **internal iterator**
    - Clients hand an internal iterator an operation to perform, and the iterator applies that operation to every element in the aggregate
- Who defines the traversal algorithm?
  - It could be defined in the iterator (making it easy to use different iteration algorithms in the same aggregate) or in the aggregate (probably necessary if we need to access private variables of the aggregate)
- Iterators for Composites
  - External iterators can be difficult to implement over recursive aggregate structures like Composites
    - They need to store a path through the Composite to keep track of the current object
- Null iterator
  - A null iterator always "is done" with the traversal
  - They can make traversing tree-structured aggregates (like Composites) easier
    - Aggregate elements return a concrete iterator as usual
    - But leaf elements return an instance of NullIterator

Operations on a compiler

- Consider a compiler that represents programs as abstract syntax trees
  - Every node is a token (a variable, an assignment, etc.)
  - The compiler needs to perform operations on the trees for various kinds of analyses
    - Type-checking
    - Code optimization
    - Flow analysis
    - …
  - Most of these operations will need to treat nodes that represent assignment statements differently from nodes that represent variables or arithmetic expressions
    - There will be one class for assignment statements, another for variable accesses, etc.
  - Distributing all our operations across the various node classes leads to a system that is hard to understand, maintain, and change
    - It will be confusing to have type-checking code mixed with pretty-printing code
  - It would be better if each new operation could be added separately, and the node classes were independent on the operations that apply to them

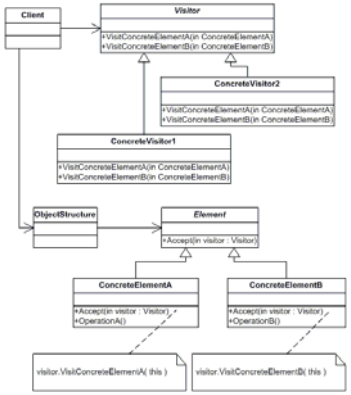
The *Visitor* pattern

- *Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates*



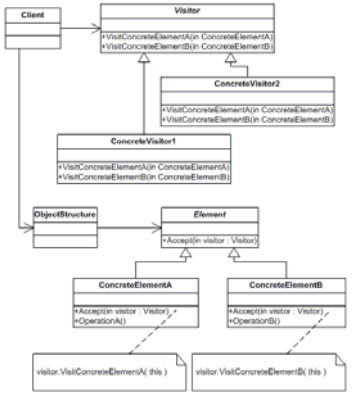
Participants and structure

- Visitor
  - Declares a Visit operation for each class of ConcreteElement in the object structure
  - The operation's name and signature identifies the class that sends the Visit request to the visitor, which lets the visitor determine the concrete class of the element being visited
- ConcreteVisitor
  - Implements each operation declared by Visitor
  - Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure



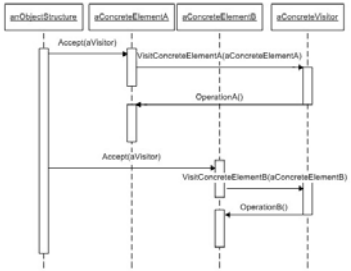
Participants and structure

- Element
  - Defines an Accept operation that takes a visitor as an argument
- ConcreteElement
  - Implements an Accept operation that takes a visitor as an argument
- ObjectStructure
  - Can enumerate its elements
  - May provide a high-level interface to allow the visitor to visit its elements
  - May either be a Composite or a collection such as a list or set



Collaborations

- A client that uses the Visitor pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor
- When the element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary



Applicability

- Use the Visitor pattern when…
  - An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes
  - Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations
  - The classes defining the object structure rarely change, but you often want to define new operations over the structure

Consequences

- Visitor makes adding new operations easy
  - You can define a new operation simply by adding a new visitor
  - In contrast, if you spread functionality over many classes, then you must change each class to define a new operation
- A visitor gathers related operations and separates unrelated ones
  - Related behavior isn't spread over the classes defining the object structure; it's localized in a visitor
  - Unrelated sets of behavior are partitioned in their own visitor classes
- Adding new ConcreteElement classes is hard
  - The Visitor pattern makes it hard to add new subclasses of Element
  - Each new ConcreteElement gives rise to a new abstract operation on Visitor and a corresponding implementation in every ConcreteVisitor class!
- Accumulating state
  - Visitors can accumulate state as they visit each element
- Breaking encapsulation
  - Visitor's approach assumes that the ConcreteElement interface is powerful enough to let visitors do their job
  - The pattern often forces you to provide public operations that access an element's internal state, which may compromise its encapsulation

Implementation

- Who is responsible for traversing the object structure?
  - A visitor must visit each element of the object structure. How does it get there?
  - We can put responsibility for traversal in any of three places: In the object structure, in the visitor, or in a separate **Iterator** object
    - Having the traversal code in the visitor is the least preferred option, as it forces you to repeat the code in every ConcreteVisitor for each ConcreteElement; so you would only follow this approach for a particularly complex traversal (e.g. one that depends on the results of the operations on the object structure)